Development of an Article Retrieval System – Report

1. Introduction

The point of this task is to build a system that indexes articles applying Retrieval Augmented Generation. To achieve that I used Python and its associated libraries and went through several key stages:

Initially, I focused on data preparation. Articles dataset was loaded into a document, which was crucial for further processing. Next, I split the text into smaller chunks to make it more manageable. Then I performed text vectorization using OpenAIEmbeddings in order to convert text into numerical data. Finally, in the next phase I implemented a search mechanism through the Chroma database that identifies and returns the most relevant article fragments. After finding the answers in the database, the model also summarizes the information that is later displayed for the user.

2. Code and used technologies

2.1. Data processing

```python
# Load the CSV file
loader = CSVLoader('medium.csv', encoding='utf-8', source_column="Text")
articles = loader.load()

# Split the text of the articles into smaller chunks
text_splitter = RecursiveCharacterTextSplitter(chunk_size=200, chunk_overlap=100)
splitted_articles = text_splitter.split_documents(articles)

# Check if the chroma path exists and delete
if os.path.isdir('chroma'):
    shutil.rmtree('chroma')

# Create database from article chunks
articles_database = Chroma.from_documents(splitted_articles, OpenAIEmbeddings(), persist_directory='chroma')
```

To carry out all necessary preprocessing and prepare data for further analysis I used functionalities from Python langchain library. To load data from csv file into the document, I used CSVLoader. Then I created RecursiveCharacterSplitter that is configured to split the text into 500 characters long chunks with an overlap of 250 characters between consecutive chunks, to ensure that no contextual information is lost. The final step was vectorization, which I performed using Chroma.from_documents function that will transform the text into a format that will be suitable for further analysis. It is doing it using OpenAIEmbeddings that are high-dimensional vectors that represents the

text. The distance and direction correspond to the similarity and relationships between the text elements. Similar words have the same embeddings, which will allow us to search for the specific information through our data.

2.2. Search mechanism

```
results = articles_database.similarity_search_with_relevance_scores(query, k)
if not results or results[0][1] < 0.7:
    return "Unable to find any information in provided database."
```

To search for the most relevant article fragments in previously built database I used similarity_search_with_relevance_scores function, where k is maximum number of returned results. This piece of code uses embeddings to find the answer to our query in the database and returns selected article chunks.

2.3. Get an answer to the query

```
context = "\n\n---\n\n".join(article.page_content for article, _ in results)
prompt = prompt_template.format(context=context, question=query)
# Create a chat model instance and get the response
chat_model = ChatOpenAI()
response = chat_model.invoke(prompt)
```

Then we should proceed to summarize the information found in previous step. Initially, I created a context to our query based on the content selected from the articles. Then context and query was put into previously declared prompt template. To get a final response I used ChatOpenAI model that summarized article pieces that were provided in a context and returned answer to the user query.

3. Summary

To sum everything up, all of these described steps were necessary to implement RAG on the provided dataset. The final code allows us to find an answer to user's question in our dataset fast an effectively. For future development, we could load more data into the Chroma database to provide more complex options and allow users to ask questions from various fields and subjects.